IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Let's practice compilation

Langages et logiques – LaLog

Objectives

At the end of the activity, you should be capable of:

- to define the syntax and implement the corresponding lexer and parser for a simple language,
- give the formal semantics of a simple language,
- implements a simple interpreter of it,
- implements a simple translation from one language to another one.

Part I
Training

Before starting the true project, we are going to start with a guided sequence. We are going to work with two languages $\mathcal{L}_1$ and $\mathcal{L}_2$. $\mathcal{L}_1$ is a simplified language to express basic boolean formula and $\mathcal{L}_2$ is a simplified low level language for a simple machine.

1 The low level language

As we do not have real hardware to execute $\mathcal{L}_2$, we are going to define an abstract machine to support the execution of a $\mathcal{L}_2$ program.
This abstract machine can be described by:

- it manipulates only one bit
- it has three registers R_{A}, R_{B} and R_{C} (of one bit)

- it has a memory of 16 bits (M_0 to M_{15}), this memory is initialized before running a program
- it supports the following operations
  - setting a register to either 0 or 1, $\mathtt{S}xb$ sets the register R_x to the value b
  - loading from memory to a register, $\mathtt{L}ix$ sets the register R_x to the value of M_i
  - computing the nand of two registers and putting the result in a register, $\mathtt{N}xyz$ puts R_x `nand` R_y in R_z where `nand` is defined by $1\,\mathtt{nand}\,1 = 0$ and $0\,\mathtt{nand}\,b = b\,\mathtt{nand}\,0 = 1$.
- a program is a sequence of the previous operations, running a program consists in executing its operation one by one until the last one

The informal semantics described above can be expressed mathematically using a small step semantics. Executing a program denoted $\mathcal{P}$ in the context of a memory $\mathcal{M}$ is done step by step by executing each of its operation while maintaining the values of the three registers denoted $\mathcal{R}$. It is expressed by a set of rules of the form $\mathcal{M} \vdash \mathcal{R}, \mathcal{P} \rightarrow \mathcal{R}', \mathcal{P}'$ where $\mathcal{R}'$ is the resulting value of the registers and $\mathcal{P}'$ the remaining program.

Accessing the value M_i in memory is done by $\mathcal{M}(i)$ (we suppose that the memory is a function from $[\![0, 15]\!]$ to $[\![0, 1]\!]$). Similarly, accessing the value of the register R_X is done by $\mathcal{R}(x)$ (we suppose that the register is a function from $\{\text{A}, \text{B}, \text{C}\}$ to $[\![0, 1]\!]$) and the register can be updated by the syntax $\mathcal{R}\{x \mapsto b\}$ meaning that $\mathcal{R}\{x \mapsto b\}(x) = b$ and if y is different from x, $\mathcal{R}\{x \mapsto b\}(y) = \mathcal{R}(y)$.

The semantics of the first operation is expressed by the rule:

$$(\mathtt{S})\ \mathcal{M} \vdash \mathcal{R}, \mathtt{S}xb\ \mathcal{P} \rightarrow \mathcal{R}\{x \mapsto b\}, \mathcal{P}$$

Exercise A

▷ **Question A.1:**
Explain this rule using plain words.

▷ **Question A.2:**
Give the rules for the semantics of load and nand.

▷ **Question A.3:**
Propose an OCaml implementation of the machine by at least defining a type `operation` for its operations and a function `step` expressing their semantics.

2 The high level language and its compilation

Our high level language allows to define simple boolean expressions containing variables. The language is defined by:

$$F ::= \mathtt{true} \mid \mathtt{false} \mid x \mid F \wedge F \mid F \vee F$$

We suppose that this language AST is implemented as follows:

```
type t =
  | Var of string
  | Bool of bool
  | And of t * t
  | Or of t * t
```

During typing, a formula containing more than 16 variables is rejected and we build a mapping from variable names to memory locations denoted M. Therefore, if we denote R the register that should contain the result (the default being $\mathsf{R_A}$), the compilation rules have the form:

$$\mathsf{M}, \mathsf{R} \vdash \text{boolean expression} \rightsquigarrow \text{instructions sequence}$$

Exercise B

▷ **Question B.1:**
Give all the compilation rules.

▷ **Question B.2:**
Give the compilation of $(\texttt{true} \vee x) \wedge (y \vee \texttt{false})$.

▷ **Question B.3:**
Propose an OCaml implementation of compilation of a formula to the one bit machine.

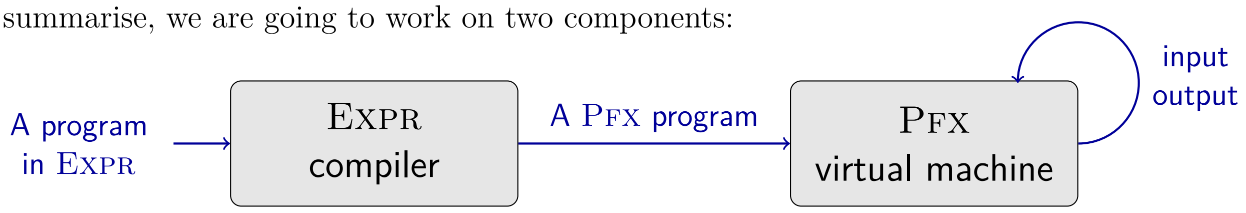
Part II
Context

3 The big picture

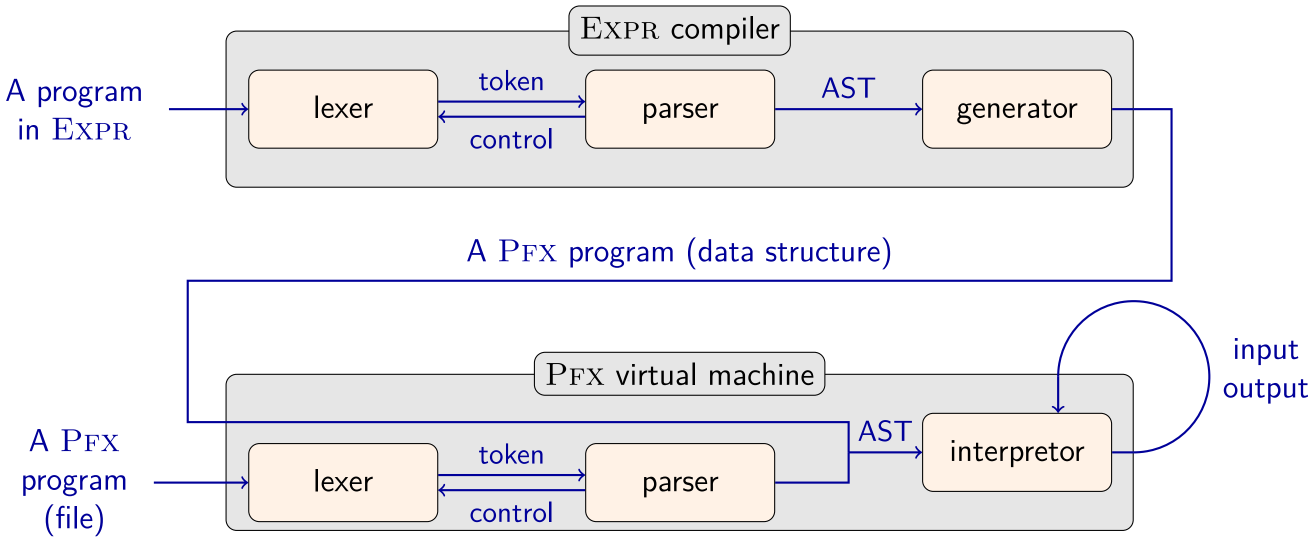
During our practice of compilation, we are going to write a compiler for a simple arithmetic expression language. This simple language is named Expr and follow directly the exercise 4 on formal calculus from the *Discovering OCaml* document.

To make it simpler for you, instead of generating X86 code to execute it on a standard machine, we are going to use a *virtual machine*. Such a virtual machine is a program that takes a low-level program and executes it. We talk about interpretation. For pedagogical reason, we stick to a simple and minimal low-level language. This low-level language is called Pfx and is a so-called stack language (it is defined later on).

To summarise, we are going to work on two components:



These two components, the Expr compiler and the Pfx virtual machine follow a similar architecture: first they parse their input language to produce an internal data structure (the AST), and then treat this AST. However, they differ in their treatment. The Expr compiler produces a corresponding Pfx program while the Pfx virtual machine *executes* its input.



For pedagogical reason and testing purpose, the Pfx virtual machine is able to receive a (Pfx) AST directly or to parse a file containing a Pfx program.

4 Work to be done

The overall objective is to produce a working Expr compiler and a working Pfx virtual machine.

The work will happen both at a mathematical (conceptual) level and at a programming level, the first being the specification of the second. You will be required to answer three kinds of questions:

- expl: you are supposed to produce a textual explanation
- math: you are supposed to give a set of mathematical definition, often in the form of a set of inference rules
- code: you are supposed to produce code; as usual, this code should be of quality (well indented, commented, with a good choice of identifiers, tested, ...)

The work is organised in a set of questions you are supposed to work on linearly. Moreover, at every step, you will have both a working Expr compiler and a working Pfx virtual machine. All along the subject, we enrich both the user language Expr and the low-level language Pfx.

During your work, you will need:

- the document containing lecture notes about *Compilation with OCaml* (`compilation-notes.pdf`). To succeed in writing your compiler, **the reading of this document is necessary**: it gives hints related to the compilation process, to the tools you will use (`ocamllex` and `menhir`). It also gives you the syntax and links to the official documentation.
- this document presenting the questions and the work to be done.

We also provide you an additional document to help you to remember the tools and commands: `OCaml_tools_cheatsheet.pdf`.

If you are proficient in OCaml and in the compilation field, just read the current document (please note that we also extend the language in order to handle simple functions and closures), then shut up and hack!

Up to the exercise you reach, all questions have to be answered. Code-related ones are answered by writing OCaml code and a quick note somewhere (comments within the code, quick summary in another file such as the README, etc.). Other questions have to be answered in a separated file, using an interoperable format (plain text or PDF for instance). It is obviously possible to answer mathematical questions by learning and using LaTeX, however this is not the point of this course (and could be time-consuming...). An efficient way to answer such questions is to scan a *readable* handwritten answer.

5 The project directory

Your project directory should be structured and this structure should be explained. We provide you with an initial template. In this template, the code is distributed in directories. Directories `expr` and `pfx` should respectively contain the code for the Expr compiler and the Pfx virtual machine. Common utilities should be defined within `utils`. Inside `expr` and `pfx`, we advocate that you use one directory for one version.

For example, in the provided template, there is a directory `basic` in the `expr` directory for the basic version of the Expr compiler. This directory corresponds to a dune library `basicExpr` defined and the `dune` file. You can use this code within `utop` by using `open BasicExpr;;`after a `dune utop`. To use it within your code, you must add `basicExpr` as a dependency within the `libraries` property of its dune file.

6 How and what to deliver?

To help you with your deliverable, we provide a project skeleton (`project_skeleton.tar.gz`) which should be renamed, adapted and extended following your needs. Please also read the README example.

We expect you to deliver a compressed archive (`.tar.gz` file[1]) containing:

- a Pfx virtual machine written in OCaml, using `ocamllex` and `menhir`, wrt the specifications that are expressed throughout the questions,
- an Expr compiler written in OCaml, using `ocamllex` and `menhir`, wrt the specifications that are expressed throughout the questions,
- a README (plain text file) explaining your work.

To be sure to be evaluated and to hope to obtain a strictly positive mark, you should also pay close attention to the form of your deliverable:

[1] `tar.gz` is not `zip`, `rar`, `7z`, `tar.xz`, `tar.bz2`, etc. and should be built using the `tar` tool.

- deliver only one compressed archive (`tar.gz format`) of your top level directory (meaning that decompressing it should result in a single directory),
- please respect the following naming convention: `NAME1-NAME2.tar.gz`,
- file encoding: UTF-8 (especially if you write non-ASCII characters…),
- clean your directory: remove all useless files and files that can be generated (for example the `_build` directory),
- choose meaningful names (file names and contents) understandable by other people,
- comment your code,
- indent your code consistently,
- organise your files (use subdirectories if needed),
- join a README file and answers to the non-code-related questions.

Part III
Questions

7 A simple stack language

During our practice of compilation, we are going to use a stack language: PFX. As expressed by its name, it is a language in the tradition of Postscript[2] and Forth[3] relying on a stack to store value instead of variables. In such a language, all operations act on the elements of the stack.

Exercise 1 (*expl*)

▷ **What is a stack? What are the operations that you usually execute on a stack?**

PFX is inspired by the Postfix machine of the book "Design Concepts in Programming Languages"[4]. It is intentionally kept simple. Whenever, you are not satisfied with it, feel free to extend and modify it… But not during the lab sessions.

7.1 Informal description

A PFX program begins with an integer specifying the number of arguments it will need to be run, then it is composed of a sequence of basic instructions. Basic instructions are `push`, `pop`, `swap` (it exchanges the first two elements of the stack) and the five arithmetic operations `add`, `sub`, `mul`, `div` and `rem`. Only `push` takes an argument which is an integer. All the arithmetic operations

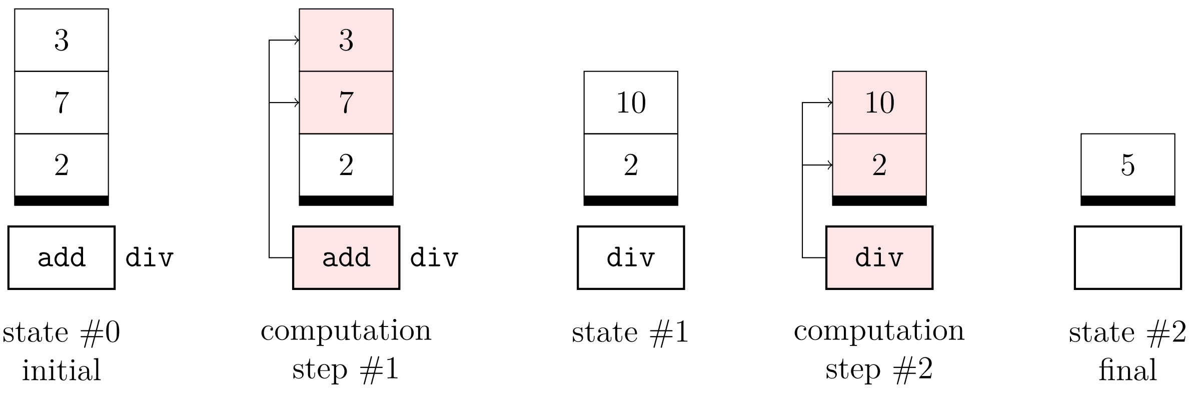
[2] `https://en.wikipedia.org/wiki/PostScript`
[3] `https://en.wikipedia.org/wiki/Forth_(programming_language)`
[4] `https://mitpress.mit.edu/books/design-concepts-programming-languages`

behave similarly: they use the first two elements of the stack as arguments, remove them and push back the result. This first version only manipulates integers so all values are integers.

For example, the program `0 push 12 push 7 sub` returns -5 while `0 push 12 push 7 swap sub` returns 5.

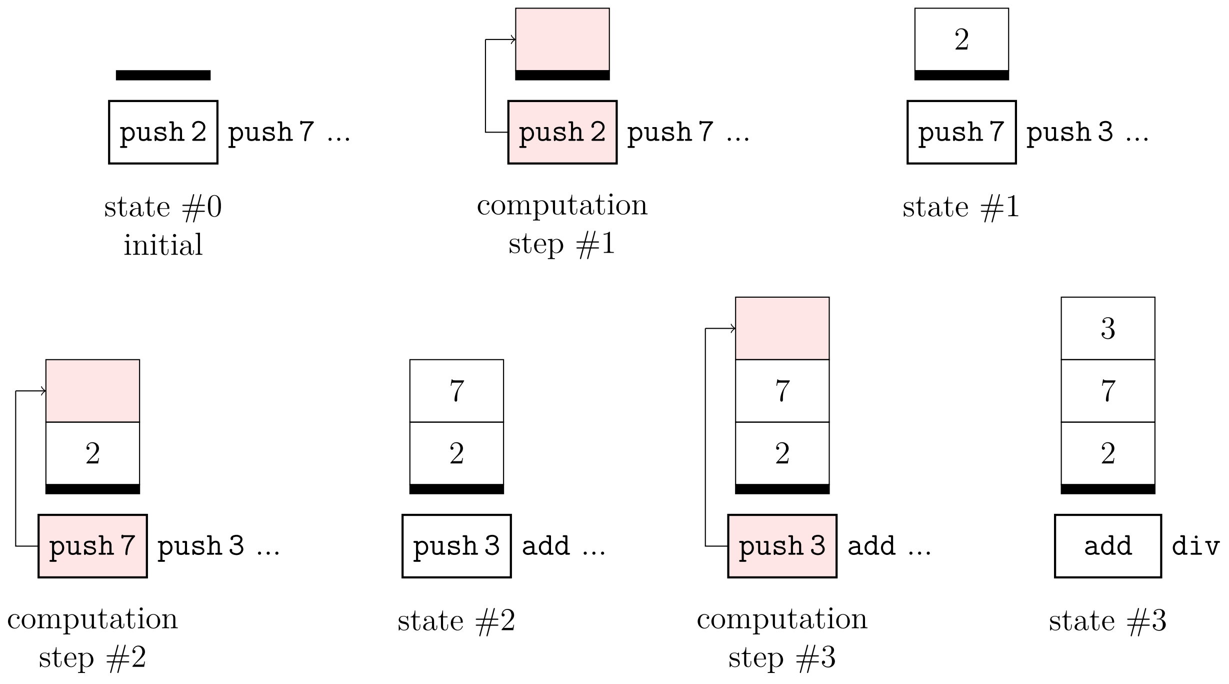
Program arguments are pushed onto the stack from last to first, before the execution of the program commands.

Here follow two examples of a program execution in detail.

> Program: `3 add div`
> Parameters: 3 7 2
>
> 
>
>
> ⇒ **result = 5**

The first example is equivalent to the following one:

Program: `0 push 2 push 7 push 3 add div`
Parameters: ∅



... as first program ...

⇒ **result** = 5

Exercise 2 (*expl*)

▷ **Detail in the same way the execution of `0 push 12 push 7 swap sub`.**

7.2 Formal semantics

To formalise the semantics of Pfx, we will use the following notations:

- a program is a pair i, Q where i is the number of awaited arguments and Q the sequence of instructions composing the program, the empty instruction sequence is written ∅,
- an instruction sequence is built by the constructor '.', the sequence composed of I then Q is $I.Q$,
- a stack is built by the constructor '::', adding an element n to the stack S is $n :: S$, the empty stack is also written ∅, the length of the stack S can be obtained by $\#S$,

The semantics of Pfx uses two sets of rules, one for programs that specify the execution of the complete program (using ⇒) and one to describe each possible computational step (using →) on a current instruction sequence and the current stack. The rules for programs use the rule for computational steps.

Exercise 3

For programs, we have the following semantics:

$$(1)\ \frac{i \neq n}{v_1, \ldots, v_n \vdash i, Q \Rightarrow \textsc{Err}} \qquad (2)\ \frac{Q, v_1 :: \ldots :: v_n :: \varnothing \rightarrow^* \textsc{Err}}{v_1, \ldots, v_n \vdash n, Q \Rightarrow \textsc{Err}}$$

$$(3)\ \frac{Q, v_1 :: \ldots :: v_n :: \varnothing \rightarrow^* \varnothing, v :: S}{v_1, \ldots, v_n \vdash n, Q \Rightarrow v}$$

The reduction rule $\rightarrow$ specifies the small step semantics of instructions and $\rightarrow^*$ its transitive closure[5]. $Q, S \rightarrow Q', S'$ means that in one step the execution of instruction sequence Q with stack S leads to instruction sequence Q' and stack S'.

▷ **Question 3.1 (*expl*):**
Explain using plain words the semantics of programs.

▷ **Question 3.2 (*math*):**
A case is still missing, spot it out and give the corresponding rule.

▷ **Question 3.3 (*math*):**
Give the rules describing the small step semantics for instruction sequences. Beware to cover all cases of runtime errors.

7.3 Implementation

Exercise 4

▷ **Question 4.1 (*code*):**
Propose the OCaml code for a type `command` describing the Pfx instructions. It should be in the file `pfx/basic/ast.ml`.

▷ **Question 4.2 (*code*):**
Write an OCaml function `step` that implements the small step reduction you defined above on Pfx instructions. It should be in the file `pfx/basic/eval.ml`.

⚠ You should test your code by running some Pfx programs defined as OCaml values (not yet parsed from files, this will be done later). You can easily do that by using `utop` and building manually building values of type `command` and test there execution calling `step`.

8 A simple arithmetic expression language

Following on the exercise 4 from the *Discovering OCaml* document, we will define a simple arithmetic expression language, named Expr in order to compile it to Pfx.

This first version of Expr AST is implemented in file `expr/basic/ast.ml` as follows:

[5]The rule is potentially applied several times in sequence. Mathematically, $A \rightarrow^* B$ if and only if $A = B$ or there exists C such that $A \rightarrow C$ and $C \rightarrow^* B$.

```
type expression =
  | Const of int
  | Var of string
  | Binop of BinOp.t * expression * expression
  | Uminus of expression
```

where the type `BinOp.t` is implemented in file `expr/common/binOp.ml` as follows:

```
type t = Badd | Bsub | Bmul | Bdiv | Bmod
```

You can explore the code provided within the project skeleton.

As described in the `README` file, you can easily test the code from `utop`. A main file able to compile a file containing an expression is also provided. It can be used by the following command from the `expr` directory:

```
dune exec ./main.exe -- basic/tests/an_example.expr
```

or by the following one from the project root:

```
dune exec expr/main.exe -- expr/basic/tests/an_example.expr
```

Exercise 5

▷ **Question 5.1 (*math*):**
Propose a compilation schema of Expr in Pfx. Give its formal description. Notice that with the current definition of Pfx, we cannot implement variables. We defer their implementation to a later exercise.

▷ **Question 5.2 (*code*):**
Define a function `generate` implementing the semantics you defined in previous question. It should be in the file `expr/basic/toPfx.ml`.

⚠ At this point, you have a first working compiler of Expr and you should be able to execute simple programs. To help, a file `compiler.ml` provides an executable that read a file given as an argument and call the translation to Pfx[6]:

```
dune exec ./compiler.exe -- basic/tests/an_example.expr
```

9 Parsing

Read the document containing the lecture notes entitled *Compilation with OCaml* (`compilation-notes.pdf` in order to understand `ocamllex` and its syntax. You can explore the code for the lexer and the parser provided within the project skeleton.

Exercise 6 (*A first* Pfx *lexer*)

▷ **Question 6.1 (*code*):**
Write a lexer for the Pfx stack machine language. Complete the provided `lexer.mll` of `pfx/basic`. To test it without the parser, have a look on Moodle at the file `simple_expr_lexer_standalone.mll` .

[6]It works as soon as you have completed your code. Before, it raises an exception `Failure("To implement")`.

The following OCaml code[7] provides a way to read the string to parse from a file. The name of the file is given as an argument on the command line and is automatically passed to the function `compile`[8].

```
let compile file =
print_string ("File "^file^" is being treated!\n");
try
 let input_file = open_in file in
 let lexbuf = Lexing.from_channel input_file in
 examine_all lexbuf;
 print_newline ();
 close_in (input_file)
with Sys_error _ ->
 print_endline ("Can't find file '" ^ file ^ "'")
let _ = Arg.parse [] compile ""
```

▷ **Question 6.2 (*code*):**
Reuse this code to be able to parse a file containing a Pfx program and prints all the tokens encountered in the process.

⚠ You should test your lexer and use test files. Now, you should use `dune` to compile and directly produce an executable file.

Exercise 7 (*Locating errors, code*)

Generally, a compiler should be able to return an error message containing the location of the error to its user. OCaml module `Lexing` defines a type `position` for this purpose.

```
type position = {
 pos_fname : string; (* name of the file *)
 pos_lnum : int;     (* number of the line *)
 pos_bol : int;      (* nb of chars between the beginning of the file and the one of current line *)
 pos_cnum : int;     (* nb of characters since the beginning of the file *)
}
```

By default, the generated lexer only updates the last element (`pos_cnum`). The actions must take care of the others. The functions `lexeme_start_p` and `lexeme_end_p` of the module `Lexing` allows one to get respectively the location of the beginning and the end of the current token. To help you, we provide the module `Location`[9] which defines helpful elements. For the moment, you should only use:

- the exception `Location.Error` carrying both a message and the location of the error;
- the type `Location.t` of a location composed of a starting position and an ending position;
- the function `Location.init` setting the file name of the given buffer;
- the function `Location.incr_line` increasing a line in the given buffer;

[7]Provided in the project skeleton, on Moodle.
[8]For more details, do not hesitate to consult the manual section on the module `Arg`.
[9]As always, available on Moodle and in the template.

- the function `Location.curr` return the current position of the given buffer;
- the function `Location.print` printing the given location.

▷ **Modify your code from the previous exercise to be able to return the location of errors.**

Exercise 8 (*A first* Pfx *parser*)

▷ **Question 8.1 (*code*):**
Write a parser for the Pfx stack machine language.

▷ **Question 8.2 (*code*):**
Test it in combination with your Lexer. To do it, you will have to write a function that prints the AST of Pfx. You should now use the provided file `pfx/pfxVM.ml` as the main file for the Pfx virtual machine. It filters every argument of the form `-a` *integer* and adds to the argument list (`args`). A `dune` file is also provided.

Notice that it requires that you modify slightly your lexer to remove the main functions and replace the token type definition by an open of the parser module.

⚠ You should test with more than one test your Pfx parser!

10 Simple functions

Let's suppose we would like to add notions of function and application to Expr. One way of doing it is to add a definition of function expression and an application expression. As in the λ-calculus, we limit ourselves to function with a unique argument. This leads to the following AST.

```
type expression =
  | Const of int
  | Var of string
  | Binop of BinOp.t * expression * expression
  | Uminus of expression
  (* For function support *)
  | App of expression * expression
  | Fun of string * expression
```

⚠ You can find the new AST and the modified lexer and parser in the directory `expr/fun` of the skeleton. You should work within this new directory. You can copy the needed files from `expr`.

Currently, Pfx cannot be used to generate code including functions. We first have to modify the language Pfx. Three new instructions must be added:

- executable sequence `(` Q `)`, where Q is an usual instruction sequence, when it is encountered the executable sequence is pushed on the top of the stack;
- `exec` which is an instruction that pops the top of the stack and executes it by appending it in front of the executing sequence, notice that the top of the stack must be an executable sequence;

- `get` pops the integer i on top of the stack, and copies on top of the stack the i-th element of the stack, it raises an error if there is not enough element on the stack.

Notice that these new constructions impose that the stack now contains two kinds of objects: integer or executable sequence.

Be sure to understand that Pfx does not have functions, it only has executable sequence.

Exercise 9

▷ **Question 9.1 (*expl*):**
Do we need to change the rules for the already defined constructs?

▷ **Question 9.2 (*math*):**
Give the formal semantics of these new constructions.

▷ **Question 9.3 (*code*):**
If needed, extend[10] the lexer and parser of Pfx to include these changes.

The translation of Expr to Pfx must be revised. The idea is that a λ-abstraction is translated to an executable sequence and an application is translated to an `exec` plus some code to *clean up* the stack. This executable sequence supposes that its parameter is on the top of the stack when it starts to execute. In its body, whenever it wants to use its parameter it `gets` it from where it is using an environment $\mathcal{P}$ associating a variable to its position in the stack (its depth). Consequently, during the translation, we have to keep track of the depth at which each parameter is. When application terminates, it pops out the parameter from the stack. This behaviour is described in the figure 1:

(a) the stack is in some state

(b) a computation pushes the argument of the future call

(c) another computation pushes the function to call, it must be an executable sequence

(d) the call is made, probably using the stack and its argument

(e) the call ends and pushes its results on the stack

Beware that during step (d), the stack may grow and therefore each time something is pushed onto the stack we need to update the position of the various reachable arguments.

Exercise 10

▷ **Question 10.1 (*expl*):**
Give the compiled version of the expression $(\lambda x.x + 1)\ 2$. Then describe step by step the evaluation of its Pfx translation.

▷ **Question 10.2 (*math*):**
Give the formal rule for transformation.

[10]Be sure to work in the `fun` directory... to keep your first version of Pfx.

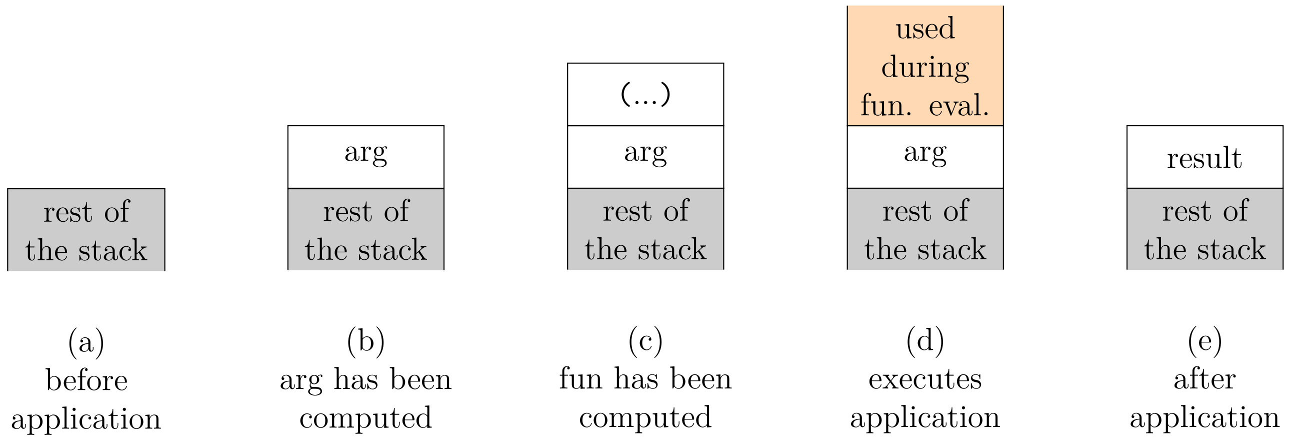


Figure 1: Usage of the stack during application

▷ **Question 10.3 (*code*):**
Provide a new version of `generate`.

▷ **Question 10.4 (*expl*):**
Give the compiled version of the expression $((\lambda x.\lambda y.(x - y))\ 12)\ 8$. Then describe step by step the evaluation of its Pfx translation. What do you think of the result? What is happening?

A straight forward extension of the syntax consists in adding a so-called *syntactic sugar* to support `let` definition (as of OCaml).

Exercise 11 (*Syntactic sugar*)

▷ **Question 11.1 (*expl*):**
What is the translation in the syntax of Expr already defined of a new `let` $x = e_1$ `in` e_2?

▷ **Question 11.2 (*code*):**
What part of the code must be modified to get support for `let`? Give the modifications.

11 Closure

To solve the problem illustrated in question 10.4, we need to change the function value. When defining a function value, we need to capture and store all its free variables (the variables defined by its context). Indeed, the function may be executed in a different scope that the one where it was defined, so it must be able to retrieve the values of variables of its definition location when executed. This new kind of value is called a *closure* and it is a pair composed of an environment (of the free variables of the function) and the function. So the closure of a function $\lambda x.e$ in the context of an environment $\mathcal{E}$ is denoted $\{\mathcal{E}, x, e\}$. Notice that the domain of $\mathcal{E}$ is limited to the set of free variables of $\lambda x.e$. When this domain is empty, we fall down on the function value of the previous section. This explains why and when it was working!

The formal semantics of Expr is presented in figure 2. The values may be integer from set $\mathbb{I}$, closures (set $\mathbb{C} = X \to (\mathbb{I} \cup \mathbb{C}) \times X \times Expr$[11]) or error (value Err). All operations are extended to produce an error whenever they are applied to an error (*e.g.* $-\text{Err} = \text{Err}$ and $\text{Err} + v = \text{Err}$).

$$(\textsc{Const})\ \mathcal{E} \vdash v \Rightarrow v \qquad (\textsc{Var})\ \frac{x \in dom(\mathcal{E})}{\mathcal{E} \vdash x \Rightarrow \mathcal{E}(x)} \qquad (\textsc{VarErr})\ \frac{x \notin dom(\mathcal{E})}{\mathcal{E} \vdash x \Rightarrow \textsc{Err}}$$

$$(\textsc{Uminus})\ \frac{\mathcal{E} \vdash e \Rightarrow v}{\mathcal{E} \vdash -e \Rightarrow -v} \qquad (\textsc{Binop})\ \frac{op \in \{+,-,*\} \qquad \mathcal{E} \vdash e_1 \Rightarrow v_1 \qquad \mathcal{E} \vdash e_2 \Rightarrow v_2}{\mathcal{E} \vdash e_1\ op\ e_2 \Rightarrow v_1\ op\ v_2}$$

$$(\textsc{Div})\ \frac{op \in \{/,\%\} \qquad \mathcal{E} \vdash e_1 \Rightarrow v_1 \qquad \mathcal{E} \vdash e_2 \Rightarrow v_2 \qquad v_2 \neq 0}{\mathcal{E} \vdash e_1\ op\ e_2 \Rightarrow v_1\ op\ v_2}$$

$$(\textsc{DivErr})\ \frac{op \in \{/,\%\} \qquad \mathcal{E} \vdash e_2 \Rightarrow 0}{\mathcal{E} \vdash e_1\ op\ e_2 \Rightarrow \textsc{Err}}$$

$$(\textsc{App})\ \frac{\mathcal{E} \vdash e_1 \Rightarrow \{\mathcal{E}_c, x, e\} \qquad \mathcal{E} \vdash e_2 \Rightarrow v_2 \qquad \mathcal{E}_c, x \mapsto v_2 \vdash e \Rightarrow v}{\mathcal{E} \vdash e_1\ e_2 \Rightarrow v}$$

$$(\textsc{AppErr1})\ \frac{\mathcal{E} \vdash e_1 \Rightarrow v_1 \qquad v_1 \notin \mathbb{C}}{\mathcal{E} \vdash e_1\ e_2 \Rightarrow v_1\ op\ v_2} \qquad (\textsc{AppErr2})\ \frac{\mathcal{E} \vdash e_1 \Rightarrow \{\mathcal{E}_c, x, e\} \qquad \mathcal{E} \vdash e_2 \Rightarrow \textsc{Err}}{\mathcal{E} \vdash e_1\ e_2 \Rightarrow \textsc{Err}}$$

$$(\textsc{Fun})\ \frac{dom(\mathcal{E}') = dom(\mathcal{E}) \qquad \forall x \in dom(\mathcal{E}), \mathcal{E}'(x) = \mathcal{E}(x)}{\mathcal{E} \vdash \lambda x.e \Rightarrow \{\mathcal{E}', x, e\}}$$

Figure 2: Big step operational semantics of Expr

Exercise 12 (*math*)

▷ **Give the proof derivation computing the value of the term of question 10.4 $(((\lambda x.\lambda y.(x-y))\ 12)\ 8)$.**

Exercise 13

▷ **Question 13.1 (*expl*):**
Is it possible to translate Expr to Pfx? If yes, can you give the idea of the translation. If no, what would be necessary to add to Pfx ?

The chosen implementation of closure we are going to explore is the following.

[11] X is the set of variables and *Expr* the set of terms of Expr.

A closure is an executable sequence that begins with instructions pushing onto the stack the values of the free variables. Notice that this part of the executable sequence must be added at runtime (the only moment when the values are known).

To enable the modification of an executable sequence at runtime we add a new construct to PFX: `append`. This construct expects the stack to contain a value on the top and an executable sequence below. It appends the command storing the "value" in the stack at the beginning of the executable sequence. When the value is an integer, this command is a push of it. The command is the value when it is an executable sequence. It also adds the operation to remove this value from the stack when the function ends.

▷ **Question 13.2 (*math*):**
Give the formal semantics of `append`.

▷ **Question 13.3 (*code*):**
If needed, extends the lexer and parser of Pfx to include these changes.

▷ **Question 13.4 (*math*):**
Give the formal rules of translation from Expr to Pfx to support closure.

▷ **Question 13.5 (*code*):**
Provide a new version of `generate`.

▷ **Question 13.6 (*expl*):**
Give the compiled version of the expression $((\lambda x.\lambda y.(x - y))\ 12)\ 8$**. Then describe step by step the evaluation of its Pfx translation. Is it better?**

12 Extensions

To explore compilation in more depth, complete the following extensions by giving the formal version and then by implementing them. For each extension, a piece of information on the difficulty is provided.

1. (*regular*) Extend the stack machine with a `sel` instruction that requires a stack of at least three elements. Its result is the third element of the stack if the first is 0 and the second in the other case. Use this extension to add booleans, boolean operators, comparison operators and a condition operator $e_1?e_2 : e_3$ similar to the one of Java. Notice that the game is to try to add as little new instructions to Pfx as possible[12].

2. (*medium*) Extend EXPR with a `let rec` construct enabling the definition of recursive functions. *Hint*[13]. Propose a mechanism to compile it to PFX. *Hint*[14].

3. (*challenging*) Extend PFX with mechanisms to manipulate the stack. In this extension, a new value can be on the stack, a stack. Add the following constructs:

[12]My solution just adds one.

[13]**Do not return before trying!**

Define a recursive function f by $\mu f.\lambda v.e$ and a recursive closure $\{\mathcal{E}, f, x, e\}$. Evaluating such a recursive closure add it to the environment under the name f.

[14]**Do not return before trying!**

The easiest implementation is to define a notion of recursive instruction sequence that is kept on the stack when executed and only popped when terminating.

- `pack` pushes a stack value containing current stack content after clearing it;
- `unpack` pops the top of the stack which must be a stack value and replaces current stack by this value;
- `switch` pops the top of the stack which must be a stack value, packs the rest of the current stack, replaces current stack by the stack value and pushes rest stack value.

These constructs provide so-called continuations. Use them to add to Expr:

(a) an exception mechanism,

(b) basic threading